ICIC Express Letters
Volume **10**, Number **12**, December **2016**

ICIC International ©2016 ISSN 1881-803X
pp. **2943–2948**

EFFICIENT LOCK ELISION FOR COARSE-GRAINED LOCK IN MULTI-THREADED PROGRAMS

Yulei Huang

ZTE Telecommunication College
Xi'an Peihua University
South Peihua Road, Chang'an District, Xi'an 710125, P. R. China
Huangyulei222@126.com

Received May 2016; accepted August 2016

Abstract. *Lock is widely used in multi-threaded programs to tune memory access from different threads. However, programming correctly with lock is not an easy task. Achieving good scalability is even harder. This paper proposes an Efficient Lock Elision Runtime System (ELERS) for coarse-grained lock in multi-threaded programs. The goal of ELERS is to achieve fine-grained performance at coarse-grained effort. Programmers can use coarse-grained lock to develop multi-threaded programs for simplicity while relying on ELERS to achieve the performance of fine-grained lock. The key idea of ELERS is to aggressively let multiple threads enter the same critical section to perform a speculative execution. By focusing on coarse-grained lock in multi-threaded programs, ELERS shows great potential to accelerate multi-threaded programs (30% speedup on red-black tree in Linux kernel, 45% speedup on splash2 benchmarks and 27% speedup over state-of-the-art lock elision system).*

Keywords: Lock elision, Coarse-grained lock, Fine-grained lock, Performance

1. **Introduction.** Multi-threaded programs have become the mainstream of applications on today's fast developing multi-core architectures. However, multi-threaded program is notoriously hard to develop. Not only is it difficult to write correct multi-threaded programs, also it is a challenge to make them scale [1]. Most multi-threaded programs use lock to synchronize threads on accessing shared data. There are two schemes of locking: coarse-grained locking and fine-grained locking. On the one hand, coarse-grained lock (e.g., using a global lock for a list) is easy to use but it generally hinders the performance and scalability. On the other hand, fine-grained lock (e.g., using a lock in each node in a list), as being designed and applied carefully, offers better performance but it is often prone to error (e.g., deadlock). Thus, a locking scheme which is at coarse-grained effort but can offer fine-grained performance is desirable.

This paper proposes an Efficient Lock Elision Runtime System (ELERS) for coarse-grained lock in multi-threaded programs. ELERS dynamically detects and optimizes coarse-grained locks in multi-threaded programs in a transparent way. In detail, for any critical section protected by coarse-grained lock, ELERS aggressively lets multiple threads enter the same critical section to perform a speculative execution. We perform conflict detection in the critical section. If we find any conflicts, some threads have to roll back.

The spirit of ELERS is similar to that of transactional memory [2] and optimistic concurrency [3]. However, certain key insights differ. Compared with transactional memory, the semantics lock offers are different from the semantics of transactional memory. Previous work [4] shows that directly transferring lock sets into transactions (atomic blocks) could introduce bugs, which is mainly because the critical sections defined by lock sets only exclude other critical sections defined by the same lock set, while in transactional memory all transactions exclude each other. Compared with previous optimistic concurrency (e.g., OPTIK) work [3], ELERS works with common lock mechanism provided

in pthread lib while previous optimistic concurrency work provides new programming interface and requires altering the source programs into another pattern.

Above all, with ELERS, we can write multi-threaded programs easily with coarse-grained locks (less programming effort) and achieve fine-grained performance (better performance). We can also optimize the performance of current legacy multi-threaded programs with ELERS in a transparent way. The key insight and contribution of ELERS is: we identify that only coarse-grained locks can be optimized into an optimistic pattern to gain performance benefit. For fine-grained locks, speculative execution would only introduce overhead (due to intense conflict of speculative execution). Based on this, we introduce a runtime system to identify coarse-grained locks and only transfer those locks into optimistic pattern. Experimental results show that compared with previous software lock elision work [5], ELERS achieves 27% speedup thanks to the precise identification of coarse-grained locks.

The rest of this paper is organized as follows. In Section 2 we introduce the problem of locking and compare it with speculative locking (our ELERS). Section 3 gives the design and implementation of ELERS. We evaluate ELERS in Section 4 and make conclusion in Section 5.

2. **The Problem of Locking.** When developing multi-threaded programs, one should determine which part of data is protected by which lock, as well as the granularity of the lock. Figure 1 shows an example of developing a simple linked list under two different schemes. Figure 1(a) shows the coarse-grained locking scheme where we use a global lock to protect the whole linked list. If a thread wants to access the linked list, it must gain the global lock first. One can easily tell that this scheme is definitely of low performance and scalability and we should avoid such design. However, such coarse-grained design widely exists in current legacy multi-threaded programs, even in the Linux kernel (for example, the Linux kernel manages the virtual space of each process using a red-black tree, which is protected by coarse-grained lock [6]). On the contrast, Figure 1(b) shows fine-grained locking scheme which is favorable. We store a lock in each node. If a thread wants to access any node, it first gains the lock for that node. Any two threads accessing different nodes can run in parallel. However, this fine-grained locking greatly complicates the program and may introduce bugs. For this simple linked list, two threads performing inserting in neighboring spaces may cause data corruption [1]. Further version-number based method should be introduced to guarantee correctness [1].

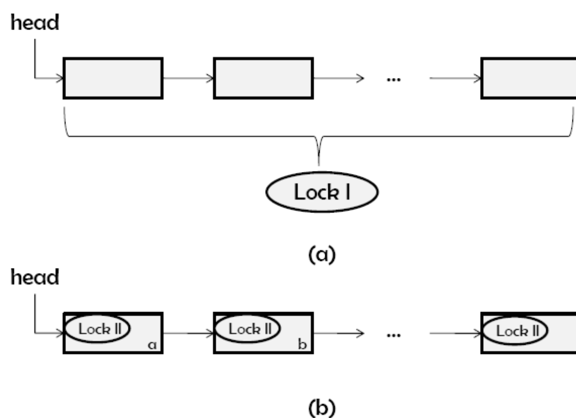

FIGURE 1. Locking schemes

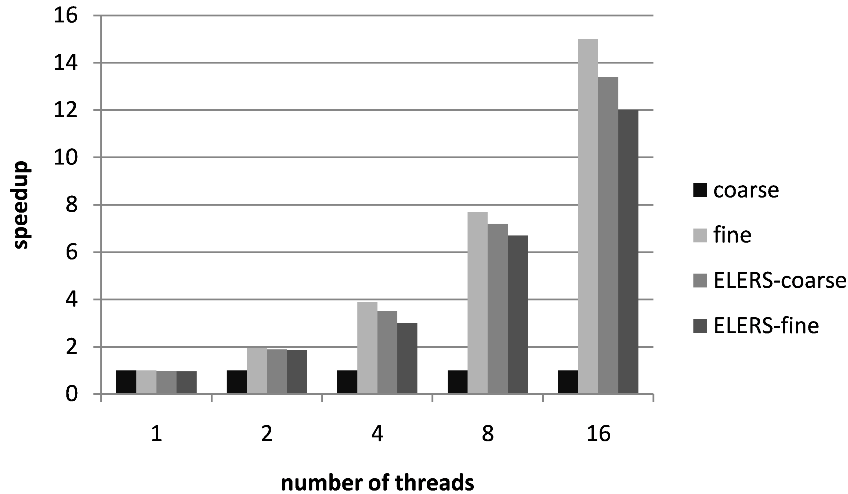

Figure 2. Performance of different schemes

Figure 2 shows the performance comparison of coarse-grained locking and fine-grained locking when doing random access-and-modification on a 1000-node list. For coarse-grained locking, it has no scalability when we increase the number of threads. On the contrast, fine-grained locking exhibits very good scalability and performance. The gap between coarse-grained locking and fine-grained locking is our opportunity. Here we also show our ELERS when it is used to optimize the coarse-grained locking (see ELERS-coarse in Figure 2) in the linked list. We can see it achieves near similar performance with fine-grained locking.

Above all, in this section we show that the simplicity of coarse-grained locking and the performance of fine-grained locking are both what we want. In Figure 2 we demonstrate that our ELERS has great potential to achieve both.

3. **Efficient Lock Elision for Multi-threaded Programs.** Figure 2 shows that our ELERS has great potential to scale up coarse-grained lock. Another important thing Figure 2 reveals is, for fine-grained locking, ELERS could introduce no benefit but only overhead (see ELERS-fine in Figure 2). This is because, in fine-grained locking scheme, parallel accesses have already been well-tuned. Making two threads speculatively enter the same critical section would most probably result into conflict. Thus in this section, we mainly introduce our mechanism of indentifying coarse-grained lock in multi-threaded programs and the whole design and implementation of our ELERS.

Speculative execution with ELERS. Similar to software transactional memory [2] system, performing speculative execution requires buffering intermediate states for each thread and committing them at the end of a critical section. In order to achieve this, we leverage the compiler framework LLVM [7] to perform static analysis on the source code and insert our own buffering code for each critical section. Generally, our code for speculative execution is based on the software transactional memory system TinySTM [2] except for that we only do conflict detection between two transactions that execute critical sections protected by the same lock (we assign each lock a special version number to achieve this).

Figure 3 shows our basic idea. We first identify each critical section (enclosed by function call to pthread_mutex_lock and pthread_mutex_unlock) during compiling time. Then we alter each memory access to a call into the TinySTM library which will record and buffer the accesses (stm_write and stm_read in Figure 3). Moreover, in order to start a transaction, we replace the original *pthread_mutex_lock* and *pthread_mutex_unlock* in pthread

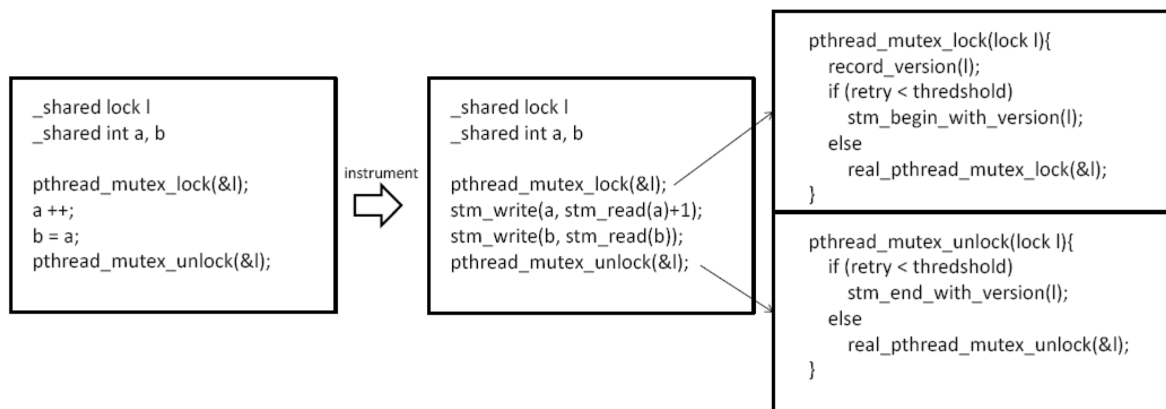

Figure 3. Overview of ELERS

library with our own implementations as shown in Figure 3. In *pthread_mutex_lock*, we first assign the transaction a version number which is derived from the lock l (the version number acts as an ID). Then we start a transaction to perform speculative execution instead of trying to get the lock. Other threads try to gain the same lock to enter critical section to perform in the same way: they first get the same version number and then execute *stm_begin_with_version(l)* to start a transaction. When a transaction ends, it just has to do conflict detection with other transactions which have the same version number (this means they want to gain the same lock). In some cases a transaction would always fail due to intense conflict. To solve this problem we added a fallback path in which all threads fall back to try to get the lock and enter the critical section serially.

Nesting lock. In a multi-threaded program there may be nested locks. As we converted lock sets into transactions, nested locks are regarded as nested transactions and run normally with TinySTM [2].

Identifying coarse-grained lock. Our final problem is how to identify a coarse-grained lock. Our method for this is based on a natural idea: **coarse-gained lock protests more data than fine-grained lock**. Threads in a critical section protected by a coarse-grained lock normally touch a wide range of memory. On the contrast, threads holding a fine-grained lock always just access a small part of memory and then release the lock. Based on this observation, our method for identifying coarse-grained lock is as follows.

(1) First, as discussed before, we have already used LLVM [7] to alter every memory accesses in each lock set into transactional function calls (stm_read and stm_write in Figure 3). Hence for each lock set at runtime, we can easily track which part of memory it protects. We track and maintain this information for each lock at runtime. For example, if a thread acquires lock l and then accesses memory a and b, we will record the information that lock l is protecting a and b. Then if another thread acquires lock l and accesses memory c and d, we will add c and d in the record.

(2) Then the whole execution of our system is described as below (see Figure 4 for detail): at the beginning of the program, the protecting information of each lock is empty. Each thread going into a critical section just tries to hold a lock and executes serially. Then in the critical section we gather the memory access information for the according lock. If we find a lock is protecting a memory region that is larger than 64 bytes (the size of a cache line), we will identify that lock as a coarse-grained lock and start speculative execution of the according critical section (as shown in Figure 4). Here that we choose the threshold to be 64 bytes is because most fine-gained locking systems organize fine-gained data that could fit into a cache line for performance reason. However, if a lock is protecting a larger memory range than this (64 bytes), then the lock is probably a coarse-grained lock.

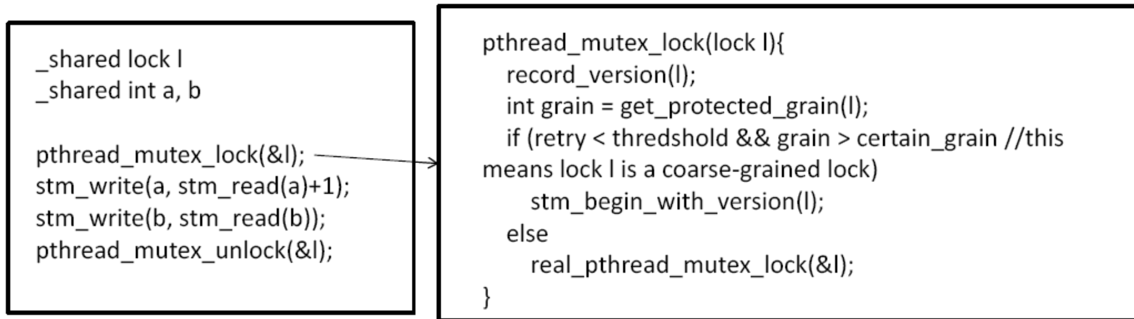

Figure 4. Focusing on coarse-grained lock

4. **Experimental Results.** We introduce a runtime system ELERS which transfers the coarse-grained lock set into optimistic transactions. In this section we mainly evaluate our ELERS on some legacy multi-threaded benchmarks to check its ability to accelerate them. We also show results on the red-black tree structure adopted in current Linux kernel [6] to show ELERS could be adopted to accelerate the kernel on managing virtual spaces of processes. For the red-black tree, we randomly insert 1000 nodes with different number of threads. Also we compare our work with previous lock elision work SLE [5] to show our advantages of just focusing on coarse-grained lock.

We conduct our experiment on a 32 core platform running Linux 3.10. The LLVM framework we use is of version 3.9.

Figure 5 shows the results. First, as shown in Figure 5(a), both our work and previous SLE achieves certain scalability on red-black tree. The coarse in Figure 5(a) means just using a global lock to protect the whole tree. Note that our ELERS performs almost the same with SLE because for coarse-grained red-black tree, there is only one lock that needs to be altered into speculative version. Thus we cannot show our advantage of identifying coarse-grained lock. However, our ELERS still achieves 30% speedup (averagely) over coarse-grained version, showing great potential to accelerate current Linux kernel.

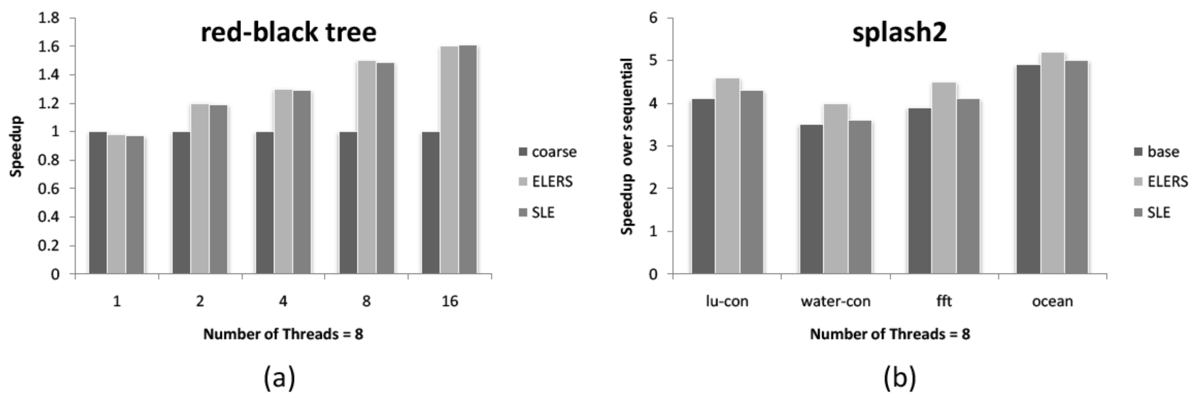

Figure 5. Experimental results

Second, Figure 5(b) shows the experimental results on benchmarks from splash2 [8] benchmark suite. Here all the benchmarks are executed with 8 threads. We show the speedup over sequential version. For all benchmarks, the original versions (base in Figure 5(b)) could achieve 3-5X speedup over sequential version. Our ELERS could further achieve 45% average speedup due to our optimization of coarse-grained lock. Compared with SLE, our ELERS could improve 27% performance thanks to our precise detection of coarse-grained lock.

Above all, the experimental results from Linux kernel red-black tree and splash2 benchmarks show that our ELERS could improve the performance of multi-threaded programs in a transparent way, showing great potential to facilitate multi-threaded programming.

5. **Conclusions.** This paper introduces an Efficient Lock Elision Runtime System (ELERS) for multi-threaded programs. ELERS aggressively lets multiple threads enter the same critical section to perform a speculative execution. By focusing on coarse-grained lock in multi-threaded programs, ELERS shows great potential to accelerate multi-threaded programs (30% speedup on red-black tree, 45% speedup on splash2 benchmarks and 27% speedup over state-of-the-art lock elision system) as well as facilitating concurrent programming. Future work will mainly be reducing the roll backs by analyzing the memory access patterns of each thread dynamically.

REFERENCES

[1] A. Matveev et al., *Read-Log-Update: A Lightweight Synchronization Mechanism for Concurrent Programming*, 2015.
[2] T. Riegel, P. Felber and C. Fetzer, *TinySTM*, 2010.
[3] R. Guerraoui and V. Trigonakis, Optimistic concurrency with OPTIK, *Proc. of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.
[4] C. Blundell, E. C. Lewis and M. Martin, Deconstructing transactional semantics: The subtleties of atomicity, *Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.
[5] A. Roy, S. Hand and T. Harris, A runtime system for software lock elision, *Proc. of the 4th ACM European Conference on Computer Systems*, 2009.
[6] A. T. Clements, M. F. Kaashoek and N. Zeldovich, RadixVM: Scalable address spaces for multithreaded applications, *Proc. of the 8th ACM European Conference on Computer Systems*, 2013.
[7] C. Lattner and V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, *IEEE International Symposium on Code Generation and Optimization*, 2004.
[8] C. Bienia, S. Kumar and K. Li, PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors, *IEEE International Symposium on Workload Characterization*, 2008.